SMART CONTRACT AUDIT REPORT

for

FWX Future Trading

Prepared By: Xiaomi Huang

PeckShield

March 28, 2023

Document Properties

Client	Forward Enterprise Limited
Title	Smart Contract Audit Report
Target	FWX Future Trading
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 28, 2023	Stephen Bie	Final Release
1.0-rc	March 8, 2023	Stephen Bie	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `FWX Future Trading` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About FWX Future Trading

`FWX Future Trading` is a decentralized derivative exchange for leveraged `Long/Short` perpetual futures. It allows traders to speculate on the direction of the price movement as well as to hedge against the risk of price fluctuation. To speculate on the price direction, one can enter a `Short` position (go short) on a futures contract written as a crypto token if he or she believes that its price will decrease. Similarly, if the price is expected to go up, one can enter a `Long` position (go long). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of FWX Future Trading

Item	Description
Name	FWX Future Trading
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 28, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that this audit only covers the `PoolBorrowing.sol`, `CoreFutureBaseFunc.sol`, `CoreFutureClosing.sol`, `CoreFutureOpening.sol`, `FeeVault.sol`, and `CoreSwapping.sol` contracts.

- https://github.com/forward-x/defi-protocol-future-trading.git (778d135)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/forward-x/defi-protocol-future-trading.git (00083b8)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `FWX Future Trading` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	3	■ ■ ■
Medium	2	■ ■
Low	0	
Informational	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 high-severity vulnerabilities and 2 medium-severity vulnerabilities.

Table 2.1: Key FWX Future Trading Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Forced Investment Risk in CoreFutureOpening::_openLong()	Business Logic	Fixed
PVE-002	Medium	Potential Sandwich/MEV Attack against CoreFutureClosing::_closeLong()	Time and State	Mitigated
PVE-003	High	Revisited Logic of CoreFutureOpening::_openLong()	Business Logic	Fixed
PVE-004	High	Public Exposure of CoreFutureOpening::openPosition()	Business Logic	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Forced Investment Risk in CoreFutureOpening::_openLong()

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `CoreFutureOpening`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [3]

Description

In the `FWX Future Trading` protocol, the `CoreFutureOpening` contract provides the operations of opening a `LONG` or `SHORT` position on a futures contract. In particular, the `_openLong()` routine is designed to open a `LONG` position. While examining its logic, we notice there is a so-called `forced investment` vulnerability that can be exploited by the malicious actor.

To elaborate, we show below the related code snippet of the `CoreFutureOpening` contract. Using the `USDT-BNB` trading pair as an example, a malicious actor supplies `$1M USDT` as collateral, and then open a `LONG` position with `10x` leverage. Inside the `_openLong()` routine, we observe `$10M USDT` is exchanged to `BNB` (line 299), which results in a huge slippage in the `Pancake USDT-BNB` pool. After that, the malicious actor performs the reverse swap to make profit.

```
283     function _openLong(
284         bytes32 pairByte,
285         uint256 tradingFee,
286         APHLibrary.OpenPositionParams memory params,
287         APHLibrary.TokenAddressParams memory addressParams
288     ) internal returns (OpenedPositionReturn memory openPos) {
289         Position memory pos = positions[params.nftId][pairByte];
290         PositionState storage posState = positionStates[params.nftId][
291             currentPositionIndex[params.nftId]
292         ];
293
294         uint256 wallet = wallets[params.nftId][pairByte];
```

```
        // swap amount is including trading fee.
        uint256 swapAmount = ((params.borrowAmount + wallet) * WEI_PERCENT_UNIT) /
            (WEI_PERCENT_UNIT + tradingFee);

        uint256[] memory amounts = _swap(
            swapAmount, // amountInMax
            params.contractSize, // amountOut
            pairByte,
            addressParams.borrowTokenAddress,
            addressParams.swapTokenAddress,
            address(this),
            true
        );

        uint256 feeAmount = _getFeeAmount(amounts[0], tradingFee);
        uint256 actualRate = (amounts[0] * WEI_UNIT) / amounts[1];

        require(
            (actualRate <= params.entryPrice
                (APHLibrary._calculateSlippage(params.entryPrice, actualRate) < params.slipPage)),
            "CoreTrading/slippage-long-too-high"
        );

        // set position
        if (!posState.active) {
            pos.contractSize -= 1;
            pos.entryPrice -= 1;
        }
        pos.entryPrice = (((pos.entryPrice * pos.contractSize) + (amounts[0] * WEI_UNIT)) /
            (pos.contractSize + amounts[1]));
        pos.contractSize = amounts[1];

        // set openPosition
        openPos.entryPrice = pos.entryPrice;
        openPos.contractSize = pos.contractSize;
        openPos.collateralSwappedAmount = (amounts[0] - params.borrowAmount);

        openPos.collaUsed = openPos.collateralSwappedAmount + feeAmount;
        openPos.feeAmount = feeAmount;
        openPos.actualRate = actualRate;
    }
```

Listing 3.1: `CoreFutureOpening::_openLong()`

Note that another routine, i.e., `_openShort()`, shares the same issue.

Recommendation Develop an effective mitigation to prevent the potential `forced investment` attack.

Status The issue has been addressed by the following commit: `5622c72`.

3.2 Potential Sandwich/MEV Attack against CoreFutureClosing::_closeLong()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CoreFutureClosing`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

Description

In the `FWX Future Trading` protocol, the `CoreFutureClosing` contract provides the operations of closing a `LONG` or `SHORT` position on a futures contract. In particular, the `_closeLong()` routine is designed to close a `LONG` position. Our analysis shows there is a potential `Sandwich/MEV` attack for the `_closeLong()` routine.

To elaborate, we show below the related code snippet of the `CoreFutureClosing` contract. Using the `USDT-BNB` trading pair as an example, when the trader closes a `LONG` position, the `_closeLong()` routine is triggered. Inside the routine, the `_swap()` routine is called (line 215) to swap a certain amount of `BNB` to `USDT`. However, we observe it essentially does not specify any restriction (with `amountOutMin=1`, line 217) on possible slippage and is therefore vulnerable to possible front-running attacks.

```
191     function _closeLong(APHLibrary.ClosePositionParams memory params)
192         internal
193         returns (APHLibrary.ClosePositionResponse memory result)
194     {
195         Position storage pos = positions[params.nftId][params.pairByte];
196         PositionState storage posState = positionStates[params.nftId][params.posId];
197         Pair memory pair = pairs[posState.pairByte];
198         PoolStat storage poolStat = poolStats[assetToPool[pair.pair0]];
199         poolStat.updatedTimestamp = block.timestamp;

201         uint256 actualCollateral = wallets[params.nftId][params.pairByte];
202         uint256 interestPaid = posState.interestPaid;

204         // swap
205         uint256[] memory amounts = params.isLiquidate
206             ? _liquidationSwapAtPancake(
207                 params.closingSize,
208                 1,
209                 params.pairByte,
210                 pos.swapTokenAddress,
211                 pos.borrowTokenAddress,
212                 address(this),
213                 false
```

```
            )
            : _swap(
                params.closingSize, // amountIn
                1, // amountOutMin
                params.pairByte,
                pos.swapTokenAddress,
                pos.borrowTokenAddress,
                address(this),
                false
            );
        ...
    }
```

Listing 3.2: `CoreFutureClosing::_closeLong()`

Note that other routines, i.e., `CoreFutureClosing::_closeShort()` and `CoreFutureBaseFunc::_getUnrealizedPNL()`, share the similar issue.

Recommendation Add necessary slippage control for above-mentioned routines to prevent possible front-running attacks.

Status The issue has been mitigated by the following commits: `5622c72` and `e808a37`.

3.3 Revisited Logic of CoreFutureOpening::_openLong()

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `CoreFutureOpening`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [3]

Description

As mentioned in Section 3.1, the `CoreFutureOpening` contract is designed to open a `LONG` or `SHORT` position on a futures contract. In particular, the `_openLong()` routine is designed to open a `LONG` position. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `CoreFutureOpening` contract. Using the `USDT-BNB` trading pair as an example, inside the `_openLong()` routine, the statement of `PositionState storage posState = positionStates[params.nftId][currentPositionIndex[params.nftId]]` is designed to retrieve the user's (specified by the `params.nftId`) `LONG` position state in the `USDT-BNB` trading pair. We observe the `currentPositionIndex[params.nftId]` is used as the position ID in the `USDT-BNB` trading pair. However, in fact, the `currentPositionIndex[params.nftId]` storage variable stores the user's latest position ID in all trading pairs rather than just the `USDT-BNB` pair. Given this, we suggest to improve the implementation as below: `PositionState storage posState = positionStates[params.nftId][pos.id]` (line 290). Note that other routines, i.e., `CoreFutureOpening::_openShort()`, `CoreFutureBaseFunc::_getPositionMargin()/_getUnrealizedPNL()`, and `CoreFutureOpening::_openPosition()`, share the similar issue.

Moreover, inside the `_openLong()` routine, the requirement of `require((actualRate <= params.entryPrice || (APHLibrary._calculateSlippage(params.entryPrice, actualRate)< params.slipPage)),"CoreTrading/slippage-long-too-high")` is executed (line 312) to ensure the user input `params.entryPrice` is larger than the actual `BNB` price (i.e., `actualRate`) or within the allowed range (i.e., `params.slipPage`). After further analysis, we notice both the `params.entryPrice` and `params.slipPage` are under the user control. A very small `params.entryPrice` can bypass the validation by providing a very large `params.slipPage`. With the unexpected `entryPrice`, the user's `PNL` will be very large, which directly undermines the assumption of the protocol design. Note that another routine, i.e., `_openShort()`, shares the similar issue.

```
283     function _openLong(
284         bytes32 pairByte,
285         uint256 tradingFee,
286         APHLibrary.OpenPositionParams memory params,
287         APHLibrary.TokenAddressParams memory addressParams
288     ) internal returns (OpenedPositionReturn memory openPos) {
289         Position memory pos = positions[params.nftId][pairByte];
290         PositionState storage posState = positionStates[params.nftId][
291             currentPositionIndex[params.nftId]
292         ];
293
294         uint256 wallet = wallets[params.nftId][pairByte];
295         // swap amount is including trading fee.
296         uint256 swapAmount = ((params.borrowAmount + wallet) * WEI_PERCENT_UNIT) /
297             (WEI_PERCENT_UNIT + tradingFee);
298
299         uint256[] memory amounts = _swap(
300             swapAmount, // amountInMax
301             params.contractSize, // amountOut
302             pairByte,
303             addressParams.borrowTokenAddress,
304             addressParams.swapTokenAddress,
305             address(this),
306             true
307         );
308
309         uint256 feeAmount = _getFeeAmount(amounts[0], tradingFee);
310         uint256 actualRate = (amounts[0] * WEI_UNIT) / amounts[1];
311
312         require(
313             (actualRate <= params.entryPrice
314                 (APHLibrary._calculateSlippage(params.entryPrice, actualRate) < params.
                    slipPage)),
315             "CoreTrading/slippage-long-too-high"
```

```
        );
        ...
    }
```

Listing 3.3: `CoreFutureOpening::_openLong()`

Recommendation Correct the implementation of above-mentioned routines.

Status The issue has been addressed by the following commit: `498cbaf`.

3.4 Public Exposure of CoreFutureOpening::openPosition()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `CoreFutureOpening`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [3]

Description

As mentioned in Section 3.1, the `CoreFutureOpening` contract provides the operations of opening a `LONG` or `SHORT` position on a futures contract. In particular, the `CoreFutureOpening::openPosition` routine is designed to meet the requirement. While examining its logic, we notice it should be properly guarded.

To elaborate, we show below the related code snippet of the contracts. By design, if the trader intends to open a `LONG` position, it should be guaranteed that his `SHORT` position in the current trading pair has been closed (line 75). We notice the related logic is implemented in the `PoolBorrowing::openPosition()` routine. However, it comes to our attention that the trader can open a `LONG` or `SHORT` position via `CoreFutureOpening::openPosition()` directly since there is no any restriction on the caller in the routine, which directly undermines the assumption of the protocol design. Given this, we suggest to limit the caller to the registered pools.

```
    function openPosition(
        APHLibrary.OpenPositionParams memory params,
        APHLibrary.TokenAddressParams memory addressParams
    ) external whenFuncNotPaused(msg.sig) nonReentrant {
        _openPosition(params, addressParams);
    }
```

Listing 3.4: `CoreFutureOpening::openPosition()`

```
45      function openPosition(
46          uint256 nftId,
47          address collateralTokenAddress,
48          address swapTokenAddress,
```

```
        uint256 entryPrice,
        uint256 _contractSize,
        uint256 leverage,
        uint256 slipPage
    ) external nonReentrant whenFuncNotPaused(msg.sig) returns (CoreBase.Position memory pos) {
        ...
        pos = _openPosition(params);
    }

    function _openPosition(APHLibrary.PoolOpenPositionParams memory poolParams)
        internal
        returns (CoreBase.Position memory pos)
    {
        uint256 nftId = _getUsableToken(msg.sender, poolParams.nftId);
        bytes32 pairByte = APHLibrary._hashPair(
            poolParams.collateralTokenAddress,
            poolParams.swapTokenAddress,
            tokenAddress
        );
        pos = IAPHCore(coreAddress).positions(nftId, pairByte);

        bool currentIsLong = pos.borrowTokenAddress == pos.collateralTokenAddress;
        bool newIsLong = tokenAddress == poolParams.collateralTokenAddress;
        uint256 contractSize = poolParams.contractSize;

        {
            if (pos.id != 0 && currentIsLong != newIsLong) {
                if ((newIsLong ? pos.borrowAmount : pos.contractSize) >= contractSize) {
                    IAPHCore(coreAddress).closePosition(nftId, pos.id, contractSize);
                    return pos;
                } else {
                    if (newIsLong) {
                        IAPHCore(coreAddress).closePosition(nftId, pos.id, pos.borrowAmount);
                        contractSize = contractSize - pos.borrowAmount;
                    } else {
                        IAPHCore(coreAddress).closePosition(nftId, pos.id, pos.contractSize);
                        contractSize = contractSize - pos.contractSize;
                    }
                }
            }
        }

        uint256 borrowAmount = newIsLong
            ? (poolParams.entryPrice * contractSize * (poolParams.leverage - WEI_UNIT)) /
                (poolParams.leverage * WEI_UNIT)
            : contractSize;

        require(
```

```
            _currentSupply() >= borrowAmount,
            "PoolBorrowing/pool-supply-sufficient-for-open-position"
        );

        (, uint256 interestOwnPerDay) = _calculateBorrowInterest(borrowAmount);

        _safeTransfer(coreAddress, borrowAmount);

        IAPHCore(coreAddress).openPosition(
            APHLibrary.OpenPositionParams(
                nftId,
                poolParams.entryPrice,
                poolParams.leverage,
                contractSize,
                poolParams.slipPage,
                borrowAmount,
                interestOwnPerDay,
                newIsLong
            ),
            APHLibrary.TokenAddressParams(
                poolParams.collateralTokenAddress, // colla
                poolParams.swapTokenAddress, // swap
                tokenAddress // borrow
            )
        );

        emit Borrow(coreAddress, nftId, tokenAddress, true, borrowAmount);
    }
```

Listing 3.5: `PoolBorrowing::openPosition()`

Recommendation Correct the `exit()` implementation by properly calculating the withdraw amount.

Status The issue has been addressed by the following commit: `ddc20a4`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `FWX Future Trading` protocol, there are a series of privileged accounts that play a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters).

In the following, we show the representative functions potentially affected by the privilege of the accounts.

```
    function setCoreFutureOpeningAddress(address _address) external onlyAddressTimelockManager {
        address oldAddress = coreFutureOpeningAddress;
        coreFutureOpeningAddress = _address;

        emit SetCoreFutureTradingAddress(msg.sender, oldAddress, _address);
    }

    function setCoreFutureClosingAddress(address _address) external onlyAddressTimelockManager {
        address oldAddress = coreFutureClosingAddress;
        coreFutureClosingAddress = _address;

        emit SetCoreFutureTradingAddress(msg.sender, oldAddress, _address);
    }
```

Listing 3.6: `CoreSetting::setCoreFutureOpeningAddress()&&setCoreFutureClosingAddress()`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. The `multi-sig` mechanism could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest to introduce the `multi-sig` mechanism to manage all the privileged accounts to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status The issue has been confirmed by the team. The team intends to introduce `timelock` mechanism to mitigate this issue.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `FWX Future Trading` protocol, which is a decentralized derivative exchange for leveraged `Long/Short` perpetual futures. It allows traders to speculate on the direction of the price movement as well as to hedge against the risk of price fluctuation. Traders can make profit via opening `Long` or `Short` position on a futures contract. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.